$P4_{16}$ Portable Switch Architecture (PSA)

(draft)

The P4.org language consortium

2017-10-21

Abstract

P4 is a language for expressing how packets are processed by the data plane of a programmable network forwarding element. P4 programs specify how the various programmable blocks of a target architecture are programmed and connected. The Portable Switch Architecture (PSA) is target architecture that describes common capabilities of network switch devices which process and forward packets across multiple interface ports.

Contents

1. Target Architecture Model

The Portable Switch Architecture (PSA) Model has six programmable P4 blocks and two fixed-function blocks, as shown in Figure 1. Programmable blocks are hardware blocks whose function can be programmed using the P4 language. The Packet buffer and Replication Engine (PRE) and the Buffer Queuing Engine (BQE) are target dependent functional blocks that may be configured for a fixed set of operations.

Incoming packets are parsed and validated, and are then passed to an ingress match action pipeline, which makes decisions on where the packets should go. After the ingress pipeline, the packet may be buffered and/or replicated (sent to multiple egress ports). For each such egress port, the packet passes through an egress match action pipeline before it is deparsed and queued to leave the pipeline. Note: the checksum operations are available – validation in the parser, and checksum computation and update in deparser (see also Table 1).

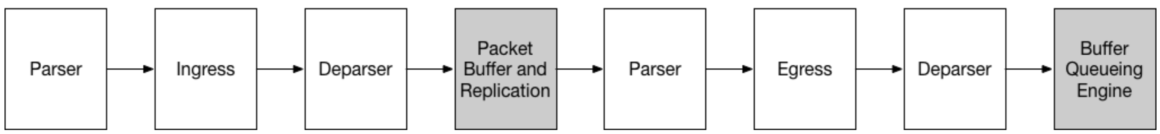

Figure 1. Portable Switch Pipeline

A programmer targeting the PSA is required to instantiate objects for the programmable blocks that conform to these APIs. Note that the programmable block APIs are templatized on user defined headers and metadata. In PSA, the user can define a single metadata type for all controls.

When instantiating the `main` `package` object, the instances corresponding to the programmable blocks are passed as arguments.

2. Naming conventions

In this document we use the following naming conventions:

- Types are named using CamelCase followed by `_t`. For example, `PortId_t`.
- Control types and extern object types are named using CamelCase. For example `IngressParser`.
- Struct types are named using lower case words separated by `_` followed by `_t`. For example `psa_ingress_input_metadata_t`.
- Actions, extern methods, extern functions, headers, structs, and instances of controls and externs start with lower case and words are separated using `_`. For example `send_to_port`.
- Enum members, const definitions, and #define constants are all caps, with words separated by `_`. For example 'PORT_CPU'.

Architecture specific metadata (e.g. structs) are prefixed by `psa_`.

3. PSA Data types

3.1. PSA type definitions

Each PSA implementation will have specific bit widths for the following types. These widths should be defined in the target specific implementation of the PSA file.

```
typedef bit<unspecified> PortId_t;
typedef bit<unspecified> MulticastGroup_t;
typedef bit<unspecified> ClassOfService_t;
```

```
typedef bit<unspecified> PacketLength_t;
typedef bit<unspecified> EgressInstance_t;
typedef bit<unspecified> Timestamp_t;

const   PortId_t         PORT_CPU = unspecified;
```

3.2. PSA supported metadata types

```
enum InstanceType_t {
    NORMAL,     /// Packet is "normal", i.e. none of the other cases below
    CLONE,      /// Packet was created via a clone operation
    RESUBMIT,   /// Packet arrival is the result of a resubmit operation
    RECIRCULATE /// Packet arrival is the result of a recirculate operation
}

struct psa_ingress_parser_input_metadata_t {
  PortId_t                 ingress_port;
  InstanceType_t           instance_type;
}

struct psa_egress_parser_input_metadata_t {
  PortId_t                 egress_port;
  InstanceType_t           instance_type;
}

struct psa_parser_output_metadata_t {
  ParserError_t            parser_error;
}

struct psa_ingress_input_metadata_t {
  // All of these values are initialized by the architecture before
  // the Ingress control block begins executing.
  PortId_t                 ingress_port;
  InstanceType_t           instance_type;
  Timestamp_t              ingress_timestamp;
  ParserError_t            parser_error;
}

struct psa_ingress_output_metadata_t {
  // The comment after each field specifies its initial value when the
  // Ingress control block begins executing.
  ClassOfService_t         class_of_service; // 0
  bool                     clone;            // false
  PortId_t                 clone_port;       // undefined
  ClassOfService_t         clone_class_of_service; // 0
  bool                     drop;             // true
  bool                     resubmit;         // false
  MulticastGroup_t         multicast_group;  // 0
  PortId_t                 egress_port;      // undefined
  bool                     truncate;         // false
  PacketLength_t           truncate_payload_bytes;  // undefined
}
```

```
struct psa_egress_input_metadata_t {
  ClassOfService_t         class_of_service;
  PortId_t                 egress_port;
  InstanceType_t           instance_type;
  EgressInstance_t         instance;       /// instance coming from PRE
  Timestamp_t              egress_timestamp;
  ParserError_t            parser_error;
}

struct psa_egress_output_metadata_t {
  // The comment after each field specifies its initial value when the
  // Egress control block begins executing.
  bool                     clone;         // false
  ClassOfService_t         clone_class_of_service; // 0
  bool                     drop;          // false
  bool                     recirculate;   // false
  bool                     truncate;      // false
  PacketLength_t           truncate_payload_bytes;  // undefined
}
```

3.3. Match kinds

Additional supported match_kind types

```
match_kind {
    range,   /// Used to represent min..max intervals
    selector /// Used for implementing dynamic_action_selection
}
```

4. PSA Externs

4.1. Restrictions on where externs may be used

All instantiations in a P4$_{16}$ program occur at compile time, and can be arranged in a tree structure we will call the instantiation tree. The root of the tree `T` represents the top level of the program. Its children are the node for the package `PSA_Switch` described in Section 5, and any externs instantiated at the top level of the program. The children of the `PSA_Switch` node are the parsers and controls passed as parameters to the `PSA_Switch` instantiation. If any of those parsers or controls instantiate other parsers, controls, and/or externs, the instantiation tree contains child nodes for them, continuing until the instantiation tree is complete.

For every instance whose node is a descendant of the `Ingress` node in this tree, call it an `Ingress` instance. Similarly for the other parameters of package `PSA_Switch`. All other instances are top level instances.

A PSA implementation is allowed to reject programs that instantiate externs, or attempt to call their methods, from anywhere other than the places mentioned in the table below.

For example, `Counter` being restricted to "Ingress, Egress" means that every `Counter` instance must be instantiated within either the Ingress control block or the Egress control block, or be a descendant of one of those nodes in the instantiation tree. If a `Counter` instance is instantiated in Ingress, for example, then it cannot be referenced, and thus its methods cannot be called, from any non-Ingress control block.

Extern type	Where it may be instantiated and called from
`ActionProfile`	Ingress, Egress
`ActionSelector`	Ingress, Egress
`Checksum`	IngressParser, EgressParser, Deparser
`Counter`	Ingress, Egress
`Digest`	Ingress, Egress
`Hash`	Ingress, Egress
`Meter`	Ingress, Egress
`DirectCounter`	Ingress, Egress
`DirectMeter`	Ingress, Egress
`Random`	Ingress, Egress
`Register`	Ingress, Egress
`ValueSet`	IngressParser, EgressParser

Table 1. Summary of controls that can instantiate and invoke externs.

PSA implementations need not support instantiating these externs at the top level. PSA implementations are allowed to accept programs that use these externs in other places, but they need not. Thus P4 programmers wishing to maximize the portability of their programs should restrict their use of these externs to the places indicated in the table.

`emit` method calls for the type `packet_out` are restricted to be within deparser control blocks in PSA, because those are the only places where an instance of type `packet_out` is visible. Similarly all methods for type `packet_in`, e.g. `extract` and `advance`, are restricted to be within parsers in PSA programs. P4$_{16}$ restricts all `verify` method calls to be within parsers, whether they are for the PSA architecture or not.

TBD: The rationale for these restrictions is: (1) it is expected that the highest performance PSA implementations will not be able to update the same extern instance from both ingress and egress, nor from more than one of the top level parsers or controls instantiated by the `PSA_Switch` package. (2) In a multi-pipeline device, there are effectively multiple instantiations of the ingress pipeline and of the egress pipeline. The primary motivation to create a multi-pipeline device is the practical difficulty in allowing the same stateful object (e.g. table, counter, etc.) to be accessed at a packet rate higher than that of a single pipeline. Thus each stateful object should be accessed from only a single pipeline on such a device.

4.2. Packet Replication Engine

The `PacketReplicationEngine` extern (abbreviated PRE) represents a part of the PSA pipeline that is not programmable via writing P4 code.

Even though the PRE can not be programmed using P4, it can be configured both directly using control plane APIs and by setting intrinsic metadata. The `psa.p4` include file provides some actions to help set these metadata fields for some common use cases, described later.

The PRE extern object has no constructor, and thus it cannot be instantiated in the user's P4 program. The architecture instantiates it exactly once, without requiring the user's P4 program to instantiate it. The PRE is made available to the Ingress programmable block using the same mechanism as `packet_in`. A corresponding Buffering and Queuing Engine (BQE) extern is defined for the Egress pipeline (see 4.3).

4.2.1. Behavior of packets after Ingress processing is complete

The pseudocode below defines where copies of packets will be made after the Ingress control block has completed executing, based upon the contents of several metadata fields in the struct `psa_ingress_output_metadata_t`.

The function `platform_port_valid()` mentioned below takes a value of type `PortId_t`, returning `true` only when the value represents an output port for the implementation. It is expected that for some PSA implementations there will be bit patterns for a value of type `PortId_t` that do not correspond to any port.

A comment saying "recommended to log error" is not a requirement, but a recommendation, that a PSA implementation should maintain a counter that counts the number of times this error occurs. It would also be useful if the implementation recorded details about the first few times this error occurred, e.g. a FIFO queue of the first several invalid values of `ostd.egress_port` that cause an error to occur, perhaps with other information about the packet that caused it, with tail dropping if it fills up. Control plane or driver software would be able to read these counters, and read and drain the FIFO queues to assist P4 developers in debugging their code.

```
struct psa_ingress_output_metadata_t {
  // The comment after each field specifies its initial value when the
  // Ingress control block begins executing.
  ClassOfService_t         class_of_service; // 0
  bool                     clone;            // false
  PortId_t                 clone_port;       // undefined
  ClassOfService_t         clone_class_of_service; // 0
  bool                     drop;             // true
  bool                     resubmit;         // false
  MulticastGroup_t         multicast_group;  // 0
  PortId_t                 egress_port;      // undefined
  bool                     truncate;         // false
  PacketLength_t           truncate_payload_bytes;  // undefined
}
```

```
    psa_ingress_output_metadata_t ostd;

    if (ostd.class_of_service value is not supported) {
        // use default class
        ostd.class_of_service = 0;
        // Recommended to log error about unsupported
        // ostd.class_of_service value.
    }
    if (truncate) {
        Truncate the payload to at most truncate_payload_bytes long.
            This affects any copies made below.
    }
    if (ostd.clone) {
        if (ostd.clone_class_of_service value is not supported) {
            ostd.clone_class_of_service = 0;
            // Recommmended to log error about unsupported
            // ostd.clone_class_of_service value.
        }
        if (platform_port_valid(ostd.clone_port)) {
            create a copy of the packet and send it to the packet
                buffer with class of service specified by
                ostd.clone_class_of_service, after which it will start
                egress processing with egress_port=ostd.clone_port (which
                might be PORT_CPU to send the clone to the control CPU).
            The contents of the packet will be the same as the packet
                as it began ingress parsing, prepended with any headers
```

```
                created by clone.emit() calls in the ingress deparser
                apply block.
        } else {
            // Do not create a clone.  Recommended to log error about
            // unsupported ostd.clone_port value.
        }
    }
    // Continue below, regardless of whether a clone was created or not.
    if (ostd.drop) {
        drop the packet
        return;   // Do not continue below.
    }
    if (ostd.resubmit) {
        resubmit the packet, i.e. it will go back to starting with the
            ingress parser;
        // TBD: Specify if anything is different about the resubmitted
        // packet vs. other copies that might be made below.
        return;   // Do not continue below.
    }
    if (ostd.multicast_group != 0) {
        Make 0 or more copies of the packet according to the control
            plane configuration of multicast group ostd.multicast_group.
            Every copy will have the same value of ostd.class_of_service
        return;   // Do not continue below.
    }
    if (platform_port_valid(ostd.egress_port)) {
        enqueue one packet for output port ostd.egress_port with class
            of service specified by ostd.class_of_service
    } else {
        drop the packet.
        // Recommended to log error about unsupported ostd.egress_port
        // value.
    }
```

TBD: Need text defining, for each possible copy, exactly what the contents of the packet will be.

TBD: Should it be possible to truncate a cloned or resubmitted packet differently than the normal packet that goes out?

TBD: If it is planned to be possible at the end of ingress to send a packet to be replicated via a multicast_group, and also have a copy go to the control CPU, give an example showing this case (after showing some simpler common cases). Ideally it should be possible for the copy going to the control CPU to have a software-defined header (defined in the P4 program) that is different than any headers on the packet copies going to the Egress control block.

A PSA implementation may implement multiple classes of service for packets sent to the packet buffer. If so, the Ingress control block may choose to assign a value to the `ostd.class_of_service` field to change the packet's class of service to a value other than the default of 0.

PSA only specifies how the Ingress control block can choose the class of service the packet is assigned. PSA does not mandate a scheduling policy among queues that exist in the packet buffer. Something at least as flexible as weighted fair queuing, with an optional strict high priority queue, is recommended.

Normally all unicast packets (i.e. those that follow the "enqueue one packet" path in the pseudocode above) received by a PSA device on the same ingress port, and sent to the same output port, will be processed by the Ingress control block in the same order they are received, and then processed by the Egress control bock in the same order as they are processed by the Ingress control block, i.e. all such packets go through the same FIFO queue in the packet buffer.

It is expected that some PSA implementations will implement the class of service mechanism by having a separate FIFO queue per class of service, and thus while unicast packets with the same ingress port, egress port, and class of service will pass through the system in FIFO order, unicast packets with the same ingress and egress port, but different classes of service, may be processed by the Egress control block in a different order than they were processed by the Ingress control block.

All of the above is also true for multicast packets (i.e. those that follow the "Make 0 or more copies" path in the pseudocode above) that are received on the same input port and their copies made to the same output port, at least during times when the multicast group memberships are stable (i.e. it has been a significant time since the control plane has updated the multicast group configuration). As for unicast packets, multicast packets with the same ingress port and replicated using the same value of `ostd.multicast_group`, but different values of `ostd.class_of_service`, are expected that they may be processed by the Egress control block in a different order than they were processed by the Ingress control block.

The control plane API excerpt below is intended to be added as part of the P4Runtime API[1].

TBD: Antonin Bas suggests exposing 'switch capabilities' type of information like that shown below not via the P4Runtime API, but instead via Openconfig data models. Ask him for a more appropriate way to document this in the PSA spec.

```
// The ClassOfServiceInfo message should be added to the "oneof"
// inside of message "Entity".

// ClassOfServiceInfo is only intended to be read.  Attempts to update
// this entity have no effect, and should return an error status that
// the entity is read only.

message ClassOfServiceInfo {
  // The number of class of service queues per output port that are
  // available in this PSA implementation.
  uint32 class_of_service_queues_per_output_port = 1;
  // The list of values of type ClassOfService_t that are supported by
  // this PSA implementation.  It is recommended that they be a
  // contiguous range from 0 up to
  // (class_of_service_queues_per_output_port - 1).
  repeated uint32 class_of_service_id = 2;
}
```

4.2.2. Behavior of packets after Egress processing is complete

The pseudocode below defines where copies of packets will be made after the Egress control block has completed executing, based upon the contents of several metadata fields in the struct `psa_egress_output_metadata_t`.

```
struct psa_egress_output_metadata_t {
  // The comment after each field specifies its initial value when the
  // Egress control block begins executing.
  bool                    clone;         // false
  ClassOfService_t        clone_class_of_service; // 0
  bool                    drop;          // false
  bool                    recirculate;   // false
  bool                    truncate;      // false
  PacketLength_t          truncate_payload_bytes;  // undefined
```

[1] The P4Runtime API, defined as a Google Protocol Buffer `.proto` file, can be found at https://github.com/p4lang/PI/blob/master/proto/p4/p4runtime.proto

```
}

    psa_egress_input_metadata_t  istd;
    psa_egress_output_metadata_t ostd;

    if (truncate) {
        Truncate the payload to at most truncate_payload_bytes long.
            This affects any copies made below.
    }
    if (ostd.clone) {
        if (ostd.clone_class_of_service value is not supported) {
            ostd.clone_class_of_service = 0;
            // Recommmended to log error about unsupported
            // ostd.clone_class_of_service value.
        }
        if (platform_port_valid(ostd.clone_port)) {
            create a copy of the packet and send it to the packet
                buffer with class of service specified by
                ostd.clone_class_of_service, after which it will start
                egress processing with egress_port=ostd.clone_port (which
                might be PORT_CPU to send the clone to the control CPU).
            The contents of the packet will be the same as the packet
                as it comes out of the egress deparser, prepended with any
                headers created by clone.emit() calls in the egress
                deparser apply block.
        } else {
            // Do not create a clone.  Recommended to log error about
            // unsupported ostd.clone_port value.
        }
    }
    // Continue below, regardless of whether a clone was created or not.
    if (ostd.drop) {
        drop the packet
        return;   // Do not continue below.
    }
    if (ostd.recirculate) {
        recirculate the packet, i.e. it will go back to starting with the
            ingress parser;
        // TBD: Specify if anything is different about the recirculated
        // packet vs. other copies that might be made below.
        return;   // Do not continue below.
    }

    // The value istd.egress_port below is the same one that the
    // packet began its Egress processing with, as decided during
    // Ingress processing for this packet.  The Egress control block
    // is not allowed to change it.
    enqueue one packet for output port istd.egress_port
```

TBD: Need text defining, for each possible copy, exactly what the contents of the packet will be, and any differences between the values of the fields in the structs `psa_parser_input_metadata_t` and `psa_egress_input_metadata_t`, in the copy, as compared to the values seen for the packet that caused those copies to be made.

TBD: Should it be possible to truncate a cloned or recirculated packet differently than the normal packet that goes out?

4.2.3. Actions for directing packets during ingress

All of these actions modify one or more metadata fields in the struct with type `psa_ingress_output_metadata_t` that is an `out` parameter of the `Ingress` control block. None of these actions has any other immediate effect. What happens to the packet is determined by the value of all fields in that struct when ingress processing is complete, not at the time one of these actions is called. See Section 4.2.1.

These actions are provided for convenience in making changes to these metadata fields. Their effects are expected to be common kinds of changes one will want to make in a P4 program. If they do not suit your use cases, you are of course welcome to modify the metadata fields directly in your P4 programs however you prefer, perhaps within actions you define yourself.

4.2.3.1. Unicast operation Sends packet to a port.

```
/// Modify ingress output metadata to cause one packet to be sent to
/// egress processing, and then to the output port egress_port, unless
/// it is dropped during egress processing.

/// This action does not change whether a clone or resubmit operation
/// will occur.

/// The one copy it causes to be sent to egress processing will have
/// its struct of type psa_egress_input_metadata_t filled in as
/// follows:

/// egress_port - equal to the egress_port parameter of this action
/// instance_type - InstanceType_t.NORMAL
/// instance - undefined
/// egress_timestamp - the time the packet begins egress processing

action send_to_port(inout psa_ingress_output_metadata_t meta,
                    in PortId_t egress_port)
{
    meta.drop = false;
    meta.multicast_group = 0;
    meta.egress_port = egress_port;
}
```

4.2.3.2. Multicast operation Sends packet to a multicast group or a port.

The multicast_group parameter is the multicast group id. The control plane must program the multicast groups through a separate mechanism.

```
/// Modify ingress output metadata to cause 0 or more copies of the
/// packet to be sent to egress processing.

/// This action does not change whether a clone or resubmit operation
/// will occur.

/// The control plane must program each multicast_group to create the
/// desired copies of the packet.  For a particular multicast group,
/// the control plane specifies a list of 0 or more copy
```

```
/// specifications:

/// (egress_port[0], instance[0]),
/// (egress_port[1], instance[1]),
/// ...,
/// (egress_port[N-1], instance[N-1])

/// Copy number i sent to egress processing will have its struct of
/// type psa_egress_input_metadata_t filled in as follows:

/// egress_port - equal to the egress_port[i]
/// instance_type - InstanceType_t.NORMAL
/// instance - instance[i]
/// egress_timestamp - the time the packet begins egress processing

action multicast(inout psa_ingress_output_metadata_t meta,
                 in MulticastGroup_t multicast_group)
{
    meta.drop = false;
    meta.multicast_group = multicast_group;
}
```

4.2.3.3. Drop operation Do not send a copy of the packet for normal egress processing.

```
/// Modify ingress output metadata to cause no packet to be sent for
/// normal egress processing.

/// This action does not change whether a clone will occur.  It will
/// prevent a packet from being resubmitted.

action ingress_drop(inout psa_ingress_output_metadata_t meta)
{
    meta.drop = true;
}
```

4.2.3.4. Truncate operation For all copies of the packet made at the end of Ingress processing, truncate the payload to be at most the specified number of bytes. Specifying 0 is legal, and causes only packet headers to be sent, with no payload.

```
/// For any copies made of this packet at the end of Ingress
/// processing, truncate the payload to at most payload_bytes bytes in
/// length.

action ingress_truncate(inout psa_ingress_output_metadata_t meta,
                        in PacketLength_t payload_bytes)
{
    meta.truncate = true;
    meta.truncate_payload_bytes = payload_bytes;
}
```

4.2.4. Clone, Recirculate, Resubmit

Figure 2 shows the proposed architectures for clone, recirculate, and resubmit in PSA.

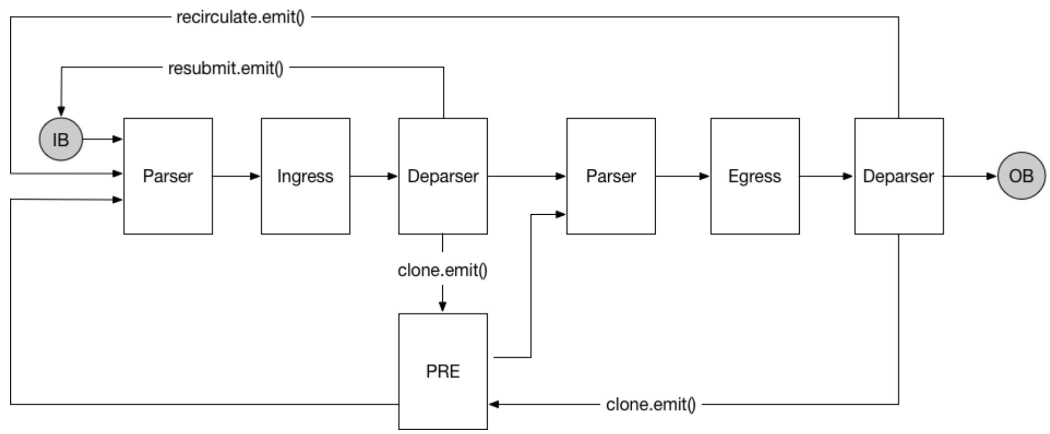

Figure 2. Clone, recirculate, and resubmit in PSA

4.2.4.1. Clone A PSA implementation provides a cloning mechanism to create copies of the original packet as independent packet instances. A cloned packet may be a duplicate of the original packet or a copy of the deparsed packet after egress pipeline. The cloning mechanism submit the cloned packet to either the ingress parser or the buffering mechanism. In a PSA implementation, the cloning mechanism is logically implemented in the PRE, that is, even if targets chose to implement it as part of other controls, the observable behavior should follow the semantics of cloning in the PRE.

The cloning mechanism can optionally attach metadata to the cloned packet. A PSA implementation provides the `clone` extern to specify the attached metadata. The `clone` extern provides a `emit` method which accepts the attached metadata of generic type `T`. In PSA, the metadata is prepended to the cloned packet. It is the responsibility of the programmer to parse the cloned packet such that it correctly extracts the attached metadata. The attached metadata may be of type `header`, `header stack`, `header union` or `struct` of the above types. Invoking the `emit` method multiple times will attach all specified metadata to the same cloned packet. The PSA architecture instantiates the `clone` extern in the ingress and egress deparser. A P4 program can use it in ingress and egress deparser only. It is an error to instantiate the `clone` extern in the P4 control or parser block.

A PSA implementation provides two configuration metadata, `clone` and `clone_port`, to the PRE to control the cloning mechanism. When `clone` is set to true, the cloning mechanism generates a cloned packet with the optional attached metadata. When `clone` is set to false, cloning is disabled and a cloned packet is not generated even if the `emit` method is invoked in the deparser. The `clone_port` controls the destination of the cloned packet. A common use case is to the send the cloned packet to the control CPU, in which case the `clone_port` should be set to the value that represents the control CPU port.

```
extern clone {
  /// Write @hdr into the ingress/egress clone engine.
  /// @T can be a header type, a header stack, a header union, or a struct
  /// containing fields with such types.
  void emit<T>(in T hdr);
}
```

4.2.4.2. Resubmit A PSA implementation provides a resubmission mechanism to resend the original packet to ingress parser for recursive processing. The resubmitted packet is the original packet as seen on the ingress pipeline. The resubmission mechanism sends the original packet to the ingress parser. The resubmission mechanism does not make copies of the original packet. In a PSA implementation, the resubmission mechanism is logically implemented in the PRE, that is, even if targets chose to implement it as part of other controls, the observable behavior should follow the semantics of resubmitting in the PRE.

The resubmission mechanism can optionally attach metadata to the resubmitted packet. A PSA implementation provides the `resubmit` extern to specify the attached metadata. The `resubmit` extern provides a `emit` method which accepts the attached metadata of generic type `T`. In a PSA implementation, the metadata is prepended to the resubmitted packet. It is the responsibility of the programmer to parse the resubmitted packet correctly to extract the attached metadata. The attached metadata can be of type `header`, `header stack`, `header union` or `struct` of the above types. Invoking the `emit` method multiple times will attach all specified metadata to the same resubmitted packet. The PSA architecture instantiates the `resubmit` extern in the ingress deparser. A P4 program can use it in ingress deparser only. It is an error to instantiate the `resubmit` extern in the P4 control or parser block.

A PSA implementation provides a configuration bit `resubmit` to the PRE to enable the resubmission mechanism. If `resubmit` is set to true, the resubmission mechanism resends the original packet with the optional attached metadata. If `resubmit` is set to false, the resubmission mechanism is disabled and the original packet is not resubmitted even if the `emit` method is invoked in the deparser.

```
extern resubmit {
  /// Write @hdr into the ingress packet buffer.
  /// @T can be a header type, a header stack, a header union or a struct
  /// containing fields with such types.
  void emit<T>(in T hdr);
}
```

4.2.4.3. Recirculate A PSA implementation provides a recirculation mechanism to send a deparsed packet from egress pipeline to ingress parser for recursive processing. The recirculated packet is the deparsed packet after the egress pipeline. The recirculation mechanism does not make copies of the original packet. It sends the deparsed packet from egress back to the ingress parser. In a PSA implementation, the recirculation mechanism is logically implemented in the PRE, that is, even if targets chose to implement it as part of other controls, the observable behavior should follow the semantics of recirculating in the PRE.

The recirculation mechanism can optionally attach metadata to the recirculated packet. A PSA implementation provides the `recirculate` extern to specify the attached metadata. The `emit` method in the `recirculate` extern accepts an attached metadata of generic type `T`. In a PSA implementation, the metadata is prepended to the recirculated packet. It is the responsibility of the programmer to parse the recirculated packet correctly to extract the attached metadata. The attached metadata can be of type `header`, `header stack`, `header union` or `struct` of the above types. Invoking the `emit` method multiple times will attach all specified metadata to the same recirculated packet. The PSA architecture instantiates the `recirculate` extern in the ingress deparser. A P4 program can use it in egress deparser only. It is an error to instantiate the `recirculate` extern in a P4 control or parser block.

A PSA implementation provides a configuration bit `recirculate` to the PRE to enable the recirculation mechanism. If `recirculate` is set to true, the recirculation mechanism sends back the deparsed packet with the optional attached metadata to the ingress parser. If `recirculate` is set to false, the recirculation mechanism is disabled and the deparsed packet is not recirculated even if the `emit` method is invoked in the deparser.

One possible implementation of the `recirculate` bit is to set the `egress_port` metadata to a dedicated recirculation port number. A PSA implementation sends the deparsed packet to the dedicated recirculation port to recirculate the packet.

A PSA implementation provides a metadata bit `is_recirc` as the input metadata to the parser to indicate if a packet is a recirculation packet. The `is_recirc` bit is true if the packet is a recirculation packet.

One possible implementation of the `is_recirc` bit is to set the bit based on the `ingress_port` metadata of the received packet. If the `ingress_port` metadata is equal to the dedicated recirculation port number, then the `is_recirc` bit is set.

```
extern recirculate {
  /// Write @hdr into the egress packet.
  /// @T can be a header type, a header stack, a header union or a struct
  /// containing fields with such types.
  void emit<T>(in T hdr);
}
```

4.3. Buffering Queuing Engine

The BufferingQueueingEngine extern (abbreviated BQE) represents another part of the PSA pipeline, after Egress, that is not programmable via writing P4 code.

Even though the BQE can not be programmed using P4, it can be configured both directly using control plane APIs and by setting intrinsic metadata.

The BQE extern object has no constructor, and thus it cannot be instantiated in the user's P4 program. The architecture instantiates it exactly once, without requiring the user's P4 program to instantiate it. The BQE is made available to the Egress programmable block using the same mechanism as packet_in. A corresponding Packet Replication Engine (PRE) extern is defined for the Ingress pipeline (see 4.2).

4.3.1. Actions for directing packets during egress

4.3.1.1. Drop operation Do not send the packet out of the device after egress processing is complete.

```
/// Modify egress output metadata to cause no packet to be sent out of
/// the device.

/// This action does not change whether a clone will occur.  It will
/// prevent a packet from being recirculated.

action egress_drop(inout psa_egress_output_metadata_t meta)
{
    meta.drop = true;
}
```

4.3.1.2. Truncate operation For all copies of the packet made at the end of Egress processing, truncate the payload to be at most the specified number of bytes. Specifying 0 is legal, and causes only packet headers to be sent, with no payload.

```
/// For any copies made of this packet at the end of Egress
/// processing, truncate the payload to at most payload_bytes bytes in
/// length.

action egress_truncate(inout psa_ingress_output_metadata_t meta,
                       in PacketLength_t payload_bytes)
{
    meta.truncate = true;
    meta.truncate_payload_bytes = payload_bytes;
}
```

4.4. Hashes

Supported hash algorithms:

```
enum HashAlgorithm_t {
  IDENTITY,
  CRC32,
  CRC32_CUSTOM,
  CRC16,
  CRC16_CUSTOM,
  ONES_COMPLEMENT16,  /// One's complement 16-bit sum used for IPv4 headers,
                      /// TCP, and UDP.
  TARGET_DEFAULT      /// target implementation defined
}
```

4.4.1. Hash function

Example usage:

```
parser P() {
  Hash<bit<16>>(HashAlgorithm_t.CRC16) h;
  bit<16> hash_value = h.get_hash(buffer);
}
```

Parameters:

- algo The algorithm to use for computation (see 4.4).
- O The type of the return value of the hash.

```
extern Hash<O> {
  /// Constructor
  Hash(HashAlgorithm_t algo);

  /// Compute the hash for data.
  /// @param data The data over which to calculate the hash.
  /// @return The hash value.
  O get_hash<D>(in D data);

  /// Compute the hash for data, with modulo by max, then add base.
  /// @param base Minimum return value.
  /// @param data The data over which to calculate the hash.
  /// @param max The hash value is divided by max to get modulo.
  ///        An implementation may limit the largest value supported,
  ///        e.g. to a value like 32, or 256.
  /// @return (base + (h % max)) where h is the hash value.
  O get_hash<T, D>(in T base, in D data, in T max);
}
```

TBD: Should there be a `const` defined that specifies the maximum allowed value of `max` parameter?

4.5. Checksums

PSA provides checksum functions compute an integer on the stream of bytes in packet headers. Checksums are often used as an integrity check to detect corrupted or otherwise malformed packets.

4.5.1. Basic checksum

The basic checksum extern provided in PSA supports arbitrary hash algorithms.
Parameters:

- W The width of the checksum

```
extern Checksum<W> {
  /// Constructor
  Checksum(HashAlgorithm_t hash);

  /// Reset internal state and prepare unit for computation
  void clear();

  /// Add data to checksum
  void update<T>(in T data);

  /// Get checksum for data added (and not removed) since last clear
  W    get();
}
```

4.5.2. Incremental checksum

PSA also provides an incremental checksum that comes equipped with an additional `remove` method that can be used to remove data previously added. The checksum is computed using the `ONES_COMPLEMENT16` hash algorithm used with protocols such as IPv4, TCP, and UDP – see IETF RFC 1624 for details.

```
// Checksum based on ‘ONES_COMPLEMENT16‘ algorithm used in IPv4, TCP, and UDP.
// Supports incremental updating via ‘remove‘ method.
// See IETF RFC 1624.
extern InternetChecksum {
  /// Constructor
  InternetChecksum();

  /// Reset internal state and prepare unit for computation
  void clear();

  /// Add data to checksum
  void update<T>(in T data);

  /// Remove data from existing checksum
  void remove<T>(in T data);

  /// Get checksum for data added (and not removed) since last clear
  bit<16>    get();
}
```

4.5.3. Checksum examples

The partial program below demonstrates one way to use the `Checksum` extern to verify whether the checksum field in a parsed IPv4 header is correct, and set a parser error if it is wrong. It also demonstrates checking for parser errors in the `ingress` control block, dropping the packet if any errors occurred during parsing. PSA programs may choose to handle packets with parser errors in other ways than shown in this example – it is up to the P4 program author to choose and write the desired behavior.

Neither $P4_{16}$ nor the PSA provide any special mechanisms to record the location within a packet that a parser error occurred. A P4 program author can choose to record such location information explicitly. For example, one may define metadata fields specifically for that purpose – e.g. to hold an encoded value representing the last parser state reached, or the number of bytes extracted so far – and then assign values to those fields within the parser state code.

```
// Define additional error values, one of them for packets with
// incorrect IPv4 header checksums.
error {
    UnhandledIPv4Options,
    BadIPv4HeaderChecksum
}

typedef bit<32> PacketCounter_t;
typedef bit<8>  ErrorIndex_t;

const bit<9> NUM_ERRORS = 256;

parser IngressParserImpl(packet_in buffer,
                         out headers parsed_hdr,
                         inout metadata user_meta,
                         in psa_ingress_parser_input_metadata_t istd,
                         out psa_parser_output_metadata_t ostd)
{
    InternetChecksum() ck;
    state start {
        buffer.extract(parsed_hdr.ethernet);
        transition select(parsed_hdr.ethernet.etherType) {
            0x0800: parse_ipv4;
            default: accept;
        }
    }
    state parse_ipv4 {
        buffer.extract(parsed_hdr.ipv4);
        // TBD: It would be good to enhance this example to
        // demonstrate checking of IPv4 header checksums for IPv4
        // headers with options, but this example does not handle such
        // packets.
        verify(parsed_hdr.ipv4.ihl == 5, error.UnhandledIPv4Options);
        ck.clear();
        ck.update({ parsed_hdr.ipv4.version,
                parsed_hdr.ipv4.ihl,
                parsed_hdr.ipv4.diffserv,
                parsed_hdr.ipv4.totalLen,
                parsed_hdr.ipv4.identification,
                parsed_hdr.ipv4.flags,
                parsed_hdr.ipv4.fragOffset,
                parsed_hdr.ipv4.ttl,
                parsed_hdr.ipv4.protocol,
                //parsed_hdr.ipv4.hdrChecksum, // intentionally leave this out
                parsed_hdr.ipv4.srcAddr,
```

```
                parsed_hdr.ipv4.dstAddr });
        // The verify statement below will cause the parser to enter
        // the reject state, and thus terminate parsing immediately,
        // if the IPv4 header checksum is wrong.  It will also record
        // the error error.BadIPv4HeaderChecksum, which will be
        // available in a metadata field in the ingress control block.
        verify(ck.get() == parsed_hdr.ipv4.hdrChecksum,
               error.BadIPv4HeaderChecksum);
        transition select(parsed_hdr.ipv4.protocol) {
            6: parse_tcp;
            default: accept;
        }
    }
    state parse_tcp {
        buffer.extract(parsed_hdr.tcp);
        transition accept;
    }
}

control ingress(inout headers hdr,
                inout metadata user_meta,
                PacketReplicationEngine pre,
                in    psa_ingress_input_metadata_t  istd,
                inout psa_ingress_output_metadata_t ostd)
{
    // Table parser_error_count_and_convert below shows one way to
    // count the number of times each parser error was encountered.
    // Although it is not used in this example program, it also shows
    // how to convert the error value into a unique bit vector value
    // 'error_idx', which can be useful if you wish to put a bit
    // vector encoding of an error into a packet header, e.g. for a
    // packet sent to the control CPU.

    DirectCounter<PacketCounter_t>(CounterType_t.PACKETS) parser_error_counts;
    ErrorIndex_t error_idx;

    action set_error_idx (ErrorIndex_t idx) {
        error_idx = idx;
        parser_error_counts.count();
    }
    table parser_error_count_and_convert {
        key = {
            istd.parser_error : exact;
        }
        actions = {
            set_error_idx;
        }
        default_action = set_error_idx(0);
        const entries = {
            error.NoError               : set_error_idx(1);
            error.PacketTooShort        : set_error_idx(2);
            error.NoMatch               : set_error_idx(3);
```

```
            error.StackOutOfBounds      : set_error_idx(4);
            error.HeaderTooShort        : set_error_idx(5);
            error.ParserTimeout         : set_error_idx(6);
            error.BadIPv4HeaderChecksum : set_error_idx(7);
            error.UnhandledIPv4Options  : set_error_idx(8);
        }
        psa_direct_counters = { parser_error_counts };
    }
    apply {
        if (istd.parser_error != error.NoError) {
            // Example code showing how to count number of times each
            // kind of parser error was seen.
            parser_error_count_and_convert.apply();
            ingress_drop(ostd);
            exit;
        }
        // Do normal packet processing here.
    }
}
```

The partial program below demonstrates one way to use the `Checksum` extern to calculate and then fill in a correct IPv4 header checksum in the deparser block. In this example, the checksum is calculated fresh, so the outgoing checksum will be correct regardless of what changes might have been made to the IPv4 header fields in the ingress (or egress) control block that precedes it.

```
control DeparserImpl(packet_out packet, inout headers hdr, in metadata meta) {
    InternetChecksum() ck;
    apply {
        ck.clear();
        ck.update({ hdr.ipv4.version,
                hdr.ipv4.ihl,
                hdr.ipv4.diffserv,
                hdr.ipv4.totalLen,
                hdr.ipv4.identification,
                hdr.ipv4.flags,
                hdr.ipv4.fragOffset,
                hdr.ipv4.ttl,
                hdr.ipv4.protocol,
                //hdr.ipv4.hdrChecksum, // intentionally leave this out
                hdr.ipv4.srcAddr,
                hdr.ipv4.dstAddr });
        hdr.ipv4.hdrChecksum = ck.get();
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);
        packet.emit(hdr.tcp);
    }
}
```

As a final example, we can use the `InternetChecksum` to compute an incremental checksum for the TCP header. Recall the TCP checksum is computed over the *entire* packet, including the payload. Because the packet payload is not available in P4, we assume that the TCP checksum on the original packet is correct, and update it incrementally by invoking `remove` and then `update` on any fields that are modified by the program. For example, the ingress control in the program below updates the IPv4 source address, recording the original source address in a metadata field:

```
control ingress(inout headers hdr,
                inout metadata user_meta,
                PacketReplicationEngine pre,
                in  psa_ingress_input_metadata_t  istd,
                out psa_ingress_output_metadata_t ostd) {
    action drop() {
      ingress_drop(ostd);
    }
    action forward(PortId_t port, bit<32> srcAddr) {
      user_meta.fwd_metadata.old_srcAddr = hdr.ipv4.srcAddr;
      hdr.ipv4.srcAddr = srcAddr;
      send_to_port(ostd, port);
    }
    table route {
        key = { hdr.ipv4.dstAddr : lpm; }
        actions = {
          forward;
          drop;
        }
    }
    apply {
        if(hdr.ipv4.isValid()) {
          route.apply();
        }
    }
}
```

The deparser first updates the IPv4 checksum as above, and then incrementally computes the TCP checksum.

```
control DeparserImpl(packet_out packet, inout headers hdr, in metadata user_meta) {
    InternetChecksum() ck;
    bit<16> hdrChecksum;
    apply {
        // Update IPv4 checksum
        ck.clear();
        ck.update({ hdr.ipv4.version,
            hdr.ipv4.ihl,
            hdr.ipv4.diffserv,
            hdr.ipv4.totalLen,
            hdr.ipv4.identification,
            hdr.ipv4.flags,
            hdr.ipv4.fragOffset,
            hdr.ipv4.ttl,
            hdr.ipv4.protocol,
            //hdr.ipv4.hdrChecksum, // intentionally leave this out
            hdr.ipv4.srcAddr,
            hdr.ipv4.dstAddr });
        hdrChecksum = ck.get();
        // Update TCP checksum
       ck.clear();
       ck.remove(hdr.tcp.checksum);
       ck.remove(user_meta.fwd_metadata.old_srcAddr);
       ck.remove(hdr.ipv4.hdrChecksum);
```

```
        ck.update(hdr.ipv4.srcAddr);
        ck.update(hdrChecksum);
        hdr.ipv4.hdrChecksum = hdrChecksum;
        hdr.tcp.checksum = ck.get();
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);
        packet.emit(hdr.tcp);
    }
}
```

4.6. Counters

Counters are a mechanism for keeping statistics. The control plane can read counter values. A P4 program cannot read counter values, only update them. If you wish to implement a feature involving sequence numbers in packets, for example, use Registers instead (Section 4.8).

Direct counters are counters associated with a particular P4 table, and are implemented by the extern `DirectCounter`. There are also indexed counters, which are implemented by the extern `Counter`. The primary differences between direct counters and indexed counters are:

- Number of independently updatable counter values:
  - A single instantiation of a direct counter always contains as many independent counter values as the number of entries in the table with which it is associated (TBD: see below for what this means for tables that use action profiles).
  - You must specify the number of independent counter values for an indexed counter when instantiating it. This number of counters need not be the same as the size of any table.
- Where counter updates are allowed in the P4 program:
  - For a direct counter, you may only invoke its `count` method from inside the actions of the table with which it is associated, and this always updates the counter value associated with the matching table entry.
  - For an indexed counter, you may invoke its `count` method anywhere in the P4 program where extern object method invocations are permitted (e.g. inside actions, or directly inside a control's `apply` block), and every such invocation must specify the index of the counter value to be updated.

Counters are only intended to support packet counters and byte counters, or a combination of both called `PACKETS_AND_BYTES`. The byte counts are always increased by some measure of the packet length, where the packet length used might vary from one PSA implementation to another. For example, one implementation might use the Ethernet frame length, including the Ethernet header and FCS bytes, as the packet arrived on a physical port. Another might not include the FCS bytes in its definition of the packet length. Another might only include the Ethernet payload length. Each PSA implementation should document how it determines the packet length used for byte counter updates.

If you wish to keep counts of other quantities, or to have more precise control over the packet length used in a byte counter, you may use Registers to achieve that (Section 4.8).

4.6.1. Counter types

```
enum CounterType_t {
    PACKETS,
    BYTES,
    PACKETS_AND_BYTES
}
```

4.6.2. Counter

```
/// Indirect counter with n_counters independent counter values, where
/// every counter value has a data plane size specified by type W.

extern Counter<W, S> {
  Counter(bit<32> n_counters, CounterType_t type);
  void count(in S index);

  /*
  /// The control plane API uses 64-bit wide counter values.  It is
  /// not intended to represent the size of counters as they are
  /// stored in the data plane.  It is expected that control plane
  /// software will periodically read the data plane counter values,
  /// and accumulate them into larger counters that are large enough
  /// to avoid reaching their maximum values for a suitably long
  /// operational time.  A 64-bit byte counter increased at maximum
  /// line rate for a 100 gigabit port would take over 46 years to
  /// wrap.

  @ControlPlaneAPI
  {
    bit<64> read      (in S index);
    bit<64> sync_read (in S index);
    void set          (in S index, in bit<64> seed);
    void reset        (in S index);
    void start        (in S index);
    void stop         (in S index);
  }
  */
}
```

See below for pseudocode of an example implementation for the Counter extern.

The example implementation for `next_counter_value` is not intended to restrict PSA implementations. In particular, the storage format for `PACKETS_AND_BYTES` type counters is just one example of how it could be done. Implementations are free to store state in other ways, as long as the control plane API returns the correct packet and byte count values.

Two common techniques for counter implementations in the data plane are:

- wrap around counters
- saturating counters, that 'stick' at their maximum possible value, without wrapping around.

This specification does not mandate any particular approach in the data plane. Implementations should strive to avoid losing information in counters. One common implementation technique is to implement an atomic "read and clear" operation in the data plane that can be invoked by the control plane software. The control plane software invokes this operation frequently enough to prevent counters from ever wrapping or saturating, and adds the values read to larger counters in driver memory.

```
Counter(bit<32> n_counters, CounterType_t type) {
    this.num_counters = n_counters;
    this.counter_vals = new array of size n_counters, each element with type W;
    this.type = type;
    if (this.type == CounterType_t.PACKETS_AND_BYTES) {
        // Packet and byte counts share storage in the same counter
```

```
        // state.  Should we have a separate constructor with an
        // additional argument indicating how many of the bits to use
        // for the byte counter?
        W shift_amount = TBD;
        this.shifted_packet_count = ((W) 1) << shift_amount;
        this.packet_count_mask = (~((W) 0)) << shift_amount;
        this.byte_count_mask = ~this.packet_count_mask;
    }
}

W next_counter_value(W cur_value, CounterType_t type) {
    if (type == CounterType_t.PACKETS) {
        return (cur_value + 1);
    }
    // Exactly which packet bytes are included in packet_len is
    // implementation-specific.
    PacketLength_t packet_len = <packet length in bytes>;
    if (type == CounterType_t.BYTES) {
        return (cur_value + packet_len);
    }
    // type must be CounterType_t.PACKETS_AND_BYTES
    // In type W, the least significant bits contain the byte
    // count, and most significant bits contain the packet count.
    // This is merely one example storage format.  Implementations
    // are free to store packets_and_byte state in other ways, as
    // long as the control plane API returns the correct separate
    // packet and byte count values.
    W next_packet_count = ((cur_value + this.shifted_packet_count) &
                           this.packet_count_mask);
    W next_byte_count = (cur_value + packet_len) & this.byte_count_mask;
    return (next_packet_count | next_byte_count);
}

void count(in S index) {
    if (index < this.num_counters) {
        this.counter_vals[index] = next_counter_value(this.counter_vals[index],
                                                      this.type);
    } else {
        // No counter_vals updated if index is out of range.
        // See below for optional debug information to record.
    }
}
```

Optional debugging information that may be kept if an `index` value is out of range includes:

- Number of times this occurs.
- A FIFO of the first N out-of-range index values that occur, where N is implementation-defined (e.g. it might only be 1). Extra information to identify which `count()` method call in the P4 program had the out-of-range `index` value is also recommended.

4.6.3. Direct Counter

```
extern DirectCounter<W> {
  DirectCounter(CounterType_t type);
  void count();

  /*
  @ControlPlaneAPI
  {
    W    read<W>      (in TableEntry key);
    W    sync_read<W> (in TableEntry key);
    void set          (in W seed);
    void reset        (in TableEntry key);
    void start        (in TableEntry key);
    void stop         (in TableEntry key);
  }
  */
}
```

A `DirectCounter` instance must appear in the list of values of the `psa_direct_counters` table attribute for exactly one table. We call this table the `DirectCounter` instance's "owner". It is an error to call the `count` method for a `DirectCounter` instance anywhere except inside an action of its owner table.

The counter value updated by an invocation of `count` is always the one associated with the table entry that matched.

TBD: How to describe which counter value is updated for tables with action profiles and direct counters? Or should this combination even be allowed?

An action of an owner table need not have `count` method calls for all of the `DirectCounter` instances that the table owns. You must use an explicit `count()` method call on a `DirectCounter` to update it, otherwise its state will not change.

An example implementation for the `DirectCounter` extern is essentially the same as the one for `Counter`. Since there is no `index` parameter to the `count` method, there is no need to check for whether it is in range.

The rules here mean that an action that calls `count` on a `DirectCounter` instance may only be an action of that instance's one owner table. If you want to have a single action `A` that can be invoked by multiple tables, you can still do so by having a unique action for each such table with a `DirectCounter`, where each such action in turn calls action `A`, in addition to any `count` invocations they have.

A `DirectCounter` instance must have a counter value associated with its owner table that is updated when there is a default action assigned to the table, and a search of the table results in a miss. If there is no default action assigned to the table, then there need not be any counter updated when a search of the table results in a miss.

By "a default action is assigned to a table", we mean that either the table has a `default_action` table property with an action assigned to it in the P4 program, or the control plane has made an explicit call to assign the table a default action. If neither of these is true, then there is no default action assigned to the table.

TBD: Verify that the method of reading this default action counter state is documented for the control plane API. I believe that Antonin Bas said that it can be accessed using the same API call used to read a `DirectCounter` value associated with a table entry, except that the key in the API call should be empty.

TBD: Should a single table be restricted to have at most one DirectCounter associated with it, or should it be allowed to have more than one?

4.6.4. Example program using counters

The following partial P4 program demonstrates the instantiation and updating of `Counter` and `DirectCounter` externs.

```
typedef bit<48> ByteCounter_t;
typedef bit<32> PacketCounter_t;
typedef bit<80> PacketByteCounter_t;

const PortId_t NUM_PORTS = 512;

struct headers {
    ethernet_t       ethernet;
    ipv4_t           ipv4;
}

control ingress(inout headers hdr,
                inout metadata user_meta,
                PacketReplicationEngine pre,
                in    psa_ingress_input_metadata_t  istd,
                inout psa_ingress_output_metadata_t ostd)
{
    Counter<ByteCounter_t, PortId_t>((bit<32>) NUM_PORTS, CounterType_t.BYTES)
        port_bytes_in;
    DirectCounter<PacketByteCounter_t>(CounterType_t.PACKETS_AND_BYTES)
        per_prefix_pkt_byte_count;

    action next_hop(PortId_t oport) {
        per_prefix_pkt_byte_count.count();
        send_to_port(ostd, oport);
    }
    action default_route_drop() {
        per_prefix_pkt_byte_count.count();
        ingress_drop(ostd);
    }
    table ipv4_da_lpm {
        key = { hdr.ipv4.dstAddr: lpm; }
        actions = {
            next_hop;
            default_route_drop;
        }
        default_action = default_route_drop;
        psa_direct_counters = {
            // table ipv4_da_lpm owns this DirectCounter instance
            per_prefix_pkt_byte_count
        };
    }
    apply {
        port_bytes_in.count(istd.ingress_port);
        if (hdr.ipv4.isValid()) {
            ipv4_da_lpm.apply();
        }
    }
```

```
}

control egress(inout headers hdr,
               inout metadata user_meta,
               BufferingQueueingEngine bqe,
               in    psa_egress_input_metadata_t  istd,
               inout psa_egress_output_metadata_t ostd)
{
    Counter<ByteCounter_t, PortId_t>((bit<32>) NUM_PORTS, CounterType_t.BYTES)
        port_bytes_out;
    apply {
        // By doing these stats updates on egress, then because
        // multicast replication happens before egress processing,
        // this update will occur once for each copy made, which in
        // this example is intentional.
        port_bytes_out.count(istd.egress_port);
    }
}
```

4.7. Meters

Meters (RFC 2698) are a more complex mechanism for keeping statistics about packets, most often used for dropping or "marking" packets that exceed an average packet or bit rate. To mark a packet means to change one or more of its quality of service values in packet headers such as the 802.1Q PCP (priority code point) or DSCP (differentiated service code point) bits within the IPv4 or IPv6 type of service byte. The meters specified in the PSA are 3-color meters.

PSA meters do not require any particular drop or marking actions, nor do they automatically implement those behaviors for you. Meters keep enough state, and update their state during `execute()` method calls, in such a way that they return a `GREEN` (also known as conform), `YELLOW` (exceed), or `RED` (violate) result. See RFC 2698 for details on the conditions under which one of these three results is returned. The P4 program is responsible for examining that returned result, and making changes to packet forwarding behavior as a result.

RFC 2698 describes "color aware" and "color blind" variations of meters. The `Meter` and `DirectMeter` externs implement both. The only difference is in which `execute` method you use when updating them. See the comments on the `extern` definitions below.

Similar to counters, there are two flavors of meters: indexed and direct. (Indexed) meters are addressed by index, while direct meters always update a meter state corresponding to the matched table entry or action, and from the control plane API are addressed using P4Runtime table entry as key.

There are many other similarities between counters and meters, including:

- The number of independently updatable meter values.
- Where meter updates are allowed in a P4 program.
- For `BYTES` type meters, the packet length used in the update is determined by the PSA implementation, and can vary from one PSA implementation to another.

Further similarities between direct counters and direct meters include:

- `DirectMeter execute` method calls must be performed within actions invoked by the table that owns the `DirectMeter` instance. It is optional for such an action to call the `execute` method.
- There must be a meter state associated with a `DirectMeter` instance's owner table, that can be updated when the table result is a miss. As for a `DirectCounter`, this state only needs to exist if a default action is assigned to the table.

The table attribute to specify that a table owns a `DirectMeter` instance is `psa_direct_meters`. The value of this table attribute is a list of meter instances.

As for counters, if you call the `execute(idx)` method on an indexed meter and `idx` is at least the number of meter states, so `idx` is out of range, no meter state is updated. The `execute` call still returns a value of type `MeterColor_t`, but the value is undefined – programs that wish to have predictable behavior across implementations must not use the undefined value in a way that affects the output packet or other side effects. The example code below shows one way to achieve predictable behavior. Note that this undefined behavior cannot occur if the value of `n_meters` of an indexed meter is 2^W, and the type `S` used to construct the meter is `bit<W>`, since the index value could never be out of range.

```
#define METER1_SIZE 100
Meter<bit<7>>(METER1_SIZE, MeterType_t.BYTES) meter1;
bit<7> idx;
MeterColor_t color1;

// ... later ...

if (idx < METER1_SIZE) {
    color1 = meter1.execute(idx, MeterColor_t.GREEN);
} else {
    // If idx is out of range, use a default value for color1.  One
    // may also choose to store an error flag in some metadata field.
    color1 = MeterColor_t.RED;
}
```

Any implementation will have a finite range that can be specified for the Peak Burst Size and Committed Burst Size. An implementation should document the maximum burst sizes they support, and if the implementation internally truncates the values that the control plane requests to something more coarse than any number of bytes, that should also be documented. It is recommended that the maximum burst sizes be allowed as large as the number of bytes that can be transmitted across the implementation's maximum speed port in 100 milliseconds.

Implementations will also have finite ranges and precisions that they support for the Peak Information Rate and Committed Information Rate. An implementation should document the maximum rate it supports, as well as the precision it supports for implementing requested rates. It is recommended that the maximum rate supported be at least the rate of the implementation's fastest port, and that the actual implemented rate should always be within plus or minus 0.1% of the requested rate.

4.7.1. Meter types

```
enum MeterType_t {
    PACKETS,
    BYTES
}
```

4.7.2. Meter colors

```
enum MeterColor_t { RED, GREEN, YELLOW };
```

4.7.3. Meter

```
// Indexed meter with n_meters independent meter states.

extern Meter<S> {
  Meter(bit<32> n_meters, MeterType_t type);

  // Use this method call to perform a color aware meter update (see
  // RFC 2698). The color of the packet before the method call was
  // made is specified by the color parameter.
  MeterColor_t execute(in S index, in MeterColor_t color);

  // Use this method call to perform a color blind meter update (see
  // RFC 2698).  It may be implemented via a call to execute(index,
  // MeterColor_t.GREEN), which has the same behavior.
  MeterColor_t execute(in S index);

  /*
  @ControlPlaneAPI
  {
    reset(in MeterColor_t color);
    setParams(in S index, in MeterConfig config);
    getParams(in S index, out MeterConfig config);
  }
  */
}
```

4.7.4. Direct Meter

```
extern DirectMeter {
  DirectMeter(MeterType_t type);
  // See the corresponding methods for extern Meter.
  MeterColor_t execute(in MeterColor_t color);
  MeterColor_t execute();

  /*
  @ControlPlaneAPI
  {
    reset(in TableEntry entry, in MeterColor_t color);
    void setConfig(in TableEntry entry, in MeterConfig config);
    void getConfig(in TableEntry entry, out MeterConfig config);
  }
  */
}
```

4.8. Registers

Registers are stateful memories whose values can be read and written during packet forwarding under the control of the P4 program. They are similar to counters and meters in that their state can be modified as a result of processing packets, but they are far more general in the behavior they can implement.

Although you may not use register contents directly in table match keys, you may use the `read()` method call on the right-hand side of an assignment statement, which retrieves the current value of the register. You may copy the register value into metadata, and it is then available for matching in subsequent tables.

A simple usage example might be to verify that a "first packet" was seen for a particular type of flow. A register cell would be allocated to the flow, initialized to "clear". When the protocol signaled a "first packet", the table would match on this value and update the flow's cell to "marked". Subsequent packets in the flow could would be mapped to the same cell; the current cell value would be stored in metadata for the packet and a subsequent table could check that the flow was marked as active.

```
extern Register<T, S> {
  Register(bit<32> size);
  T    read  (in S index);
  void write (in S index, in T value);

  /*
  @ControlPlaneAPI
  {
    T    read<T>      (in S index);
    void set          (in S index, in T seed);
    void reset        (in S index);
  }
  */
}
```

Another example using registers is given below. It implements a packet and byte counter, where the byte counter can be updated by a packet length specified in the P4 program, rather than one chosen by the PSA implementation.

```
const PortId_t NUM_PORTS = 512;

// It would be more convenient to use a struct type to represent the
// state of a combined packet and byte count, and many other compound
// values one might wish to store in a Register instance.  However,
// the latest p4test as of 2017-Aug-13 does not allow a struct type to
// be returned from a method call like Register.read().

#define PACKET_COUNT_WIDTH 32
#define BYTE_COUNT_WIDTH 48
//#define PACKET_BYTE_COUNT_WIDTH (PACKET_COUNT_WIDTH + BYTE_COUNT_WIDTH)
#define PACKET_BYTE_COUNT_WIDTH 80

#define PACKET_COUNT_RANGE (PACKET_BYTE_COUNT_WIDTH-1):BYTE_COUNT_WIDTH
#define BYTE_COUNT_RANGE (BYTE_COUNT_WIDTH-1):0

typedef bit<PACKET_BYTE_COUNT_WIDTH> PacketByteCountState_t;

action update_pkt_ip_byte_count (inout PacketByteCountState_t s,
                                 in bit<16> ip_length_bytes)
{
    s[PACKET_COUNT_RANGE] = s[PACKET_COUNT_RANGE] + 1;
    s[BYTE_COUNT_RANGE] = (s[BYTE_COUNT_RANGE] +
                           (bit<BYTE_COUNT_WIDTH>) ip_length_bytes);
}
```

```
control ingress(inout headers hdr,
                inout metadata user_meta,
                PacketReplicationEngine pre,
                in    psa_ingress_input_metadata_t  istd,
                inout psa_ingress_output_metadata_t ostd)
{
    Register<PacketByteCountState_t, PortId_t>((bit<32>) NUM_PORTS)
        port_pkt_ip_bytes_in;

    apply {
        ostd.egress_port = 0;
        if (hdr.ipv4.isValid()) {
            @atomic {
                PacketByteCountState_t tmp;
                tmp = port_pkt_ip_bytes_in.read(istd.ingress_port);
                update_pkt_ip_byte_count(tmp, hdr.ipv4.totalLen);
                port_pkt_ip_bytes_in.write(istd.ingress_port, tmp);
            }
        }
    }
}
```

Note the use of the `@atomic` annotation in the block enclosing the `read()` and `write()` method calls on the `Register` instance. It is expected to be common that register accesses will need the `@atomic` annotation around portions of your program in order to behave as you desire. As stated in the P4_16 specification, without the `@atomic` annotation in this example, an implementation is allowed to process two packets `P1` and `P2` in parallel, and perform the register access operations in this order:

```
    // Possible order of operations for the example program if the
    // @atomic annotation is _not_ used.

    tmp = port_pkt_ip_bytes_in.read(istd.ingress_port);  // for packet P1
    tmp = port_pkt_ip_bytes_in.read(istd.ingress_port);  // for packet P2

    // At this time, if P1 and P2 came from the same ingress_port,
    // each of their values of tmp are identical.

    update_pkt_ip_byte_count(tmp, hdr.ipv4.totalLen);    // for packet P1
    update_pkt_ip_byte_count(tmp, hdr.ipv4.totalLen);    // for packet P2

    port_pkt_ip_bytes_in.write(istd.ingress_port, tmp);  // for packet P1
    port_pkt_ip_bytes_in.write(istd.ingress_port, tmp);  // for packet P2
    // The write() from packet P1 is lost.
```

Since different implementations may have different upper limits on the complexity of code that they will accept within an `@atomic` block, we recommend you keep them as small as possible, subject to maintaining your desired correct behavior.

Individual counter and meter method calls need not be enclosed in `@atomic` blocks to be safe – they guarantee atomic behavior of their individual method calls, without losing any updates.

As for indexed counters and meters, access to an index of a register that is at least the size of the register is out of bounds. An out of bounds write has no effect on the state of the system. An out of bounds read returns an undefined value. See the example in Section 4.7 for one way to write code to guarantee avoiding this undefined behavior. Out of bounds register accesses are impossible for a register instance with type S declared as bit<W> and size 2^W entries.

4.9. Random

The Random extern provides generation of pseudo-random numbers in a specified range with a uniform distribution. If one wishes to generate numbers with a non-uniform distribution, you may do so by first generating a uniformly distributed random value, and then using appropriate table lookups and/or arithmetic on the resulting value to achieve the desired distribution.

An implementation is not required to produce cryptographically strong pseudo-random number generation. For example, a particularly inexpensive implementation might use a linear feedback shift register to generate values.

```
extern Random<T> {
  Random(T min, T max);
  T read();

  /*
  @ControlPlaneAPI
  {
    void reset();
    void setSeed(in T seed);
  }
  */
}
```

4.10. Action Profile

Action profiles are used as table implementation attributes.

Action profiles provide a mechanism to populate table entries with action specifications that have been defined outside the table entry specification. An action profile extern can be instantiated as a resource in the P4 program. A table that uses this action profile must specify its implementation attribute as the action profile instance.

Table entry	Key (h.f: lpm)	Action spec.
t1	01001*	set_port(1)
t2	1100*	set_port(2)
t3	101*	set_port(1)

(a) Direct table.

Table entry	Key (h.f: lpm)	Member ref.
t1	01001*	m1
t2	1100*	m2
t3	101*	m1

Member ref.	Action spec.
m1	set_port(1)
m2	set_port(2)

(b) Indirect table with action profile implementation.

Figure 3. Action profiles in PSA

Figure 3 contrasts a direct table with a table that has an action profile implementation. A direct table, as seen in Figure 3 (a) contains the action specification in each table entry. In this example, the table has a match key consisting of an LPM on header field `h.f`. The action is to set the port. As we can see, entries t1 and t3 have the same action, i.e. to set the port to 1. Action profiles enable sharing an action across multiple entries by using a separate table as shown in Figure 3 (b).

A table with an action profile implementation has entries that point to a member reference instead of directly defining an action specification. A mapping from member references to action specifications is maintained in a separate table that is part of the action profile instance defined in the table implementation attribute. When a table with an action profile implementation is applied, the member reference is resolved and the corresponding action specification is applied to the packet.

Action profile members may only specify action types defined in the `actions` attribute of the implemented table. An action profile instance may be shared across multiple tables only if all such tables define the same set of actions in their `actions` attribute. Tables with an action profile implementation cannot define a default action. The default action for such tables is implicitly set to `NoAction`.

The control plane can add, modify or delete member entries for a given action profile instance. The controller-assigned member reference must be unique in the scope of the action profile instance. An action profile instance may hold at most `size` entries as defined in the constructor parameter. Table entries must specify the action using the controller-assigned reference for the desired member entry. Directly specifying the action as part of the table entry is not allowed for tables with an action profile implementation.

```
extern ActionProfile {
  /// Construct an action profile of 'size' entries
  ActionProfile(bit<32> size);

  /*
  @ControlPlaneAPI
  {
     entry_handle add_member    (action_ref, action_data);
     void         delete_member (entry_handle);
     entry_handle modify_member (entry_handle, action_ref, action_data);
  }
  */
}
```

4.10.1. Action Profile Example

The P4 control block `Ctrl` in the example below instantiates an action profile `ap` that can contain at most 128 member entries. Table `indirect` uses this instance by specifying the implementation attribute. The control plane can add member entries to `ap`, where each member can specify either a `foo` or `NoAction` action. Table entries for `indirect` table must specify the action using the controller-assigned member reference.

```
control Ctrl(inout H hdr, inout M meta) {

  action foo() { meta.foo = 1; }

  action_profile ap(32w128);

  table indirect {
    key = {hdr.ipv4.dst_address: exact;}
    actions = { foo; NoAction; }
    implementation = ap;
  }
```

```
  apply {
    indirect.apply();
  }
}
```

4.11. Action Selector

Action selectors are used as table implementation attributes.

Action selectors implement yet another mechanism to populate table entries with action specifications that have been defined outside the table entry. They are more powerful than action profiles because they also provide the ability to dynamically select the action specification to apply upon maching a table entry. An action selector extern can be instantiated as a resource in the P4 program, similar to action profiles. Furthermore, a table that uses this action selector must specify its implementation attribute as the action selector instance.

Table entry	Key (h.f: lpm)	Member/ Group ref.
t1	01001*	g1
t2	1100*	m2
t3	101*	g2

Group ref.	Members
g1	m1, m2
g2	m1
g3	m2

Member ref.	Action spec.
m1	set_port(1)
m2	set_port(2)

Figure 4. Action selectors in PSA

Figure 4 illustrates a table that has an action selector implementation. In this example, the table has a match key consisting of an LPM on header field `h.f`. A second match type `selector` is used to define the fields that are used to look up the action specification from the selector at runtime.

A table with an action action selector implementation consists of entries that point to either an action profile member reference or an action profile group reference. An action selector instance can be logically visualized as two tables as shown in Figure 4. The first table contains a mapping from group references to a set of member references. The second table contains a mapping from member references to action specifications.

When a packet matches a table entry at runtime, the controller-assigned reference of the action profile member or group is read. If the entry points to a member then the corresponding action specification is applied to the packet. However, if the entry points to a group, a dynamic selection algorithm is used to select a member from the group, and the action specification corresponding to that member is applied. The dynamic selection algorithm is specified as a parameter when instantiating the action selector.

Action selector members may only specify action types defined in the `actions` attribute of the implemented table. All actions in a group must be of the same type. The action parameters for actions in the same group are allowed to differ, and the action of different groups in a selector may be different. An action selector instance may be shared across multiple tables only if all such tables define the same set of actions in their `actions` attribute. Furthermore, the selector match fields for such tables must be identical and must be specified in the same order across all tables sharing the selector. Tables with an action selector implementation cannot define a default action. The default action for such tables is implicitly set to `NoAction`.

The dynamic selection algorithm requires a field list as an input for generating the index to a member entry in a group. This field list is created by using the match type `selector` when defining the table match key. The match fields of type `selector` are composed into a field list in the order they are specified. The composed field list is passed as an input to the action selector implementation. It is illegal to define a `selector` type match field if the table does not have an action selector implemenation.

The control plane can add, modify or delete member and group entries for a given action selector instance. An action selector instance may hold atmost `size` member entries as defined in the constructor parameter. The number of groups may be atmost the size of the table that is implemented by the selector. Table entries must specify the action using a reference to the desired member or group entry. Directly specifying the action as part of the table entry is not allowed for tables with an action selector implementation.

```
extern ActionSelector {
  /// Construct an action selector of 'size' entries
  /// @param algo hash algorithm to select a member in a group
  /// @param size number of entries in the action selector
  /// @param outputWidth size of the key
  ActionSelector(HashAlgorithm_t algo, bit<32> size, bit<32> outputWidth);

  /*
  @ControlPlaneAPI
  {
     entry_handle add_member        (action_ref, action_data);
     void         delete_member     (entry_handle);
     entry_handle modify_member     (entry_handle, action_ref, action_data);
     group_handle create_group      ();
     void         delete_group      (group_handle);
     void         add_to_group      (group_handle, entry_handle);
     void         delete_from_group (group_handle, entry_handle);
  }
  */
}
```

4.11.1. Action Selector Example

The P4 control block `Ctrl` in the example below instantiates an action selector `as` that can contain at most 128 member entries. The action selector uses a crc16 algorithm with output width of 10 bits to select a member entry within a group.

Table `indirect_with_selection` uses this instance by specifying the implementation attribute as shown. The control plane can add member and group entries to `as`. Each member can specify either a `foo` or `NoAction` action. When programming the table entries, the control plane *does not* include the fields of match type `selector` in the match key. The selector match fields are instead used to compose a list that is passed to the action selector instance. In the example below, the list {hdr.ipv4.src_address, hdr.ipv4.protocol} is passed as input to the crc16 hash algorithm used for dynamic member selection by action selector `as`.

```
control Ctrl(inout H hdr, inout M meta) {

  action foo() { meta.foo = 1; }

  action_selector as(HashAlgorithm.crc16, 32w128, 32w10);

  table indirect_with_selection {
    key = {
      hdr.ipv4.dst_address: exact;
      hdr.ipv4.src_address: selector;
      hdr.ipv4.protocol: selector;
    }
    actions = { foo; NoAction; }
```

```
    implementation = as;
  }

  apply {
    indirect_with_selection.apply();
  }
}
```

4.12. Parser Value Sets

A parser value set is a named set of values that may be used during packet header parsing time to make decisions. You may use control plane API calls to add values to a set, and remove values from a set, at run time, much like P4 tables. Unlike tables, they may not have actions associated with them. They may only be used to determine whether a particular value is in the set, returning a Boolean value. That Boolean value can then be used in a `select` statement to control parsing (see examples below).

```
extern ValueSet<D> {
    ValueSet(int<32> size);
    bool is_member(in D data);

    /*
    @ControlPlaneAPI
    message ValueSetEntry {
        uint32 value_set_id = 1;
        // FieldMatch allows specification of exact, lpm, ternary, and
        // range matching on fields for tables, and these options are
        // permitted for the ValueSet extern as well.
        repeated FieldMatch match = 2;
    }

    // ValueSetEntry should be added to the 'message Entity'
    // definition, inside its 'oneof Entity' list of possibilities.
    */
}
```

The control plane API excerpt above is intended to be added as part of the P4Runtime API[1].

The control plane API for a `ValueSet` is similar to that of a table, except only match fields may be specified, with no actions. This includes API calls that specify ternary or range matching, although for `ValueSet`s these do not require specifying any priority values, since the only result of a `ValueSet` `is_member` call is "in the set" or "not in the set".

If a PSA target can do so, it should implement control plane API calls involving ternary or range matching using ternary or range matching capabilities in the target, consuming the minimal table entries possible.

However, a PSA target is allowed to implement such control plane API calls by "expanding" them into as many exact match entries as needed to have the same behavior. For example, a control plane API call adding all values in the range 5 through 8 may be implemented as adding the four separate exact match values 5, 6, 7, and 8.

The parser definition below shows an example that uses two `ValueSet` instances called `tpid_types` and `trill_types`.

```
parser IngressParserImpl(packet_in buffer,
```

[1]The P4Runtime API, defined as a Google Protocol Buffer `.proto` file, can be found at https://github.com/p4lang/PI/blob/master/proto/p4/p4runtime.proto

```
                        out headers parsed_hdr,
                        inout metadata user_meta,
                        in psa_ingress_parser_input_metadata_t istd,
                        out psa_parser_output_metadata_t ostd)
{
    ValueSet<bit<16>>(4) tpid_types;
    ValueSet<bit<16>>(2) trill_types;
    state start {
        buffer.extract(parsed_hdr.ethernet);
        transition select(parsed_hdr.ethernet.etherType) {
            0x0800: parse_ipv4;
            0x86DD: parse_ipv6;
            default: dispatch_tpid_value_set;
        }
    }
    state dispatch_tpid_value_set {
        bool is_tpid = tpid_types.is_member(parsed_hdr.ethernet.etherType);
        transition select(is_tpid) {
            true: parse_vlan_tag;
            default: dispatch_trill_value_set;
        }
    }
    state dispatch_trill_value_set {
        bool is_trill = trill_types.is_member(parsed_hdr.ethernet.etherType);
        transition select(is_trill) {
            true: parse_trill;
            default: accept;
        }
    }
    state parse_vlan_tag {
        // extract VLAN 802.1Q header here
        transition accept;
    }
    state parse_trill {
        // extract TRILL header here
        transition accept;
    }
    state parse_ipv4 {
        transition accept;
    }
    state parse_ipv6 {
        transition accept;
    }
}
```

The second example (below) has the same parsing behavior as the example above, but combines the two parse states dispatch_tpid_value_set and dispatch_trill_value_set into one.

```
    state dispatch_tpid_value_set {
        bool is_tpid = tpid_types.is_member(parsed_hdr.ethernet.etherType);
        bool is_trill = trill_types.is_member(parsed_hdr.ethernet.etherType);
        transition select(is_tpid, is_trill) {
            (true,     _): parse_vlan_tag;
            (false, true): parse_trill;
```

```
            default: accept;
        }
    }
```

The third example (below) demonstrates one way to have a `ValueSet` that matches on multiple fields, by making the type `D` a `struct` containing multiple bit vectors.

```
struct CustomValueSet1_t {
    bit<16> etherType;
    bit<8>  partialMacAddress;
}

parser IngressParserImpl(packet_in buffer,
                         out headers parsed_hdr,
                         inout metadata user_meta,
                         in psa_ingress_parser_input_metadata_t istd,
                         out psa_parser_output_metadata_t ostd)
{
    ValueSet<CustomValueSet1_t>(2) trill_types;

    state dispatch_tpid_value_set {
        bool is_trill =
            trill_types.is_member({parsed_hdr.ethernet.etherType,
                                   parsed_hdr.ethernet.dstAddr[7:0]});
        transition select(is_trill) {
            true: parse_vlan_tag;
            default: accept;
        }
    }

    // ... etc.
```

A PSA compliant implementation is not required to support any use of a `ValueSet` `is_member` method call return value, other than directly inside of a `select` expression. For example, a program fragment like the one shown below may be rejected, and thus P4 programmers striving for maximum portability should avoid writing such code.

```
    bool is_tpid = tpid_types.is_member(parsed_hdr.ethernet.etherType);

    is_tpid = is_tpid && (parsed_hdr.ethernet.dstAddr[47:40] == 0xfe);
    transition select(is_tpid) {
        // ...
```

4.13. Timestamps

A PSA implementation provides an `ingress_timestamp` value for every packet in the ingress control block, as a field in the struct with type `psa_ingress_input_metadata_t`. This timestamp should be close to the time that the first bit of the packet arrived to the device, or alternately, to the time that the device began parsing the packet. This timestamp is *not* automatically included with the packet in the egress control block. A P4 program wishing to use the value of `ingress_timestamp` in egress code must copy it to a user-defined metadata field that reaches egress.

A PSA implementation also provides an `egress_timestamp` value for every packet in the egress control block, as a field of the struct with type `psa_egress_input_metadata_t`.

One expected use case for timestamps is to store them in tables or `Register` instances to implement checking for timeout events for protocols, where precision on the order of milliseconds is sufficient for most protocols.

Another expected use case is INT (Inband Network Telemetry[2]), where precision on the order of microseconds or smaller is necessary to measure queueing latencies that differ by those amounts. It takes only 0.74 microseconds to transmit a 9 Kbyte Ethernet jumbo frame on a 100 gigabit per second link.

For these applications, it is recommended that an implementation's timestamp increments at least once every microsecond. Incrementing once per clock cycle in an ASIC or FPGA implementation would be a reasonable choice. The timestamp should increment at a constant rate over time. For example, it should not be a simple count of clock cycles in a device that implements dynamic frequency scaling[3].

Timestamps are of type `Timestamp_t`, which is type `bit<W>` for a value of `W` defined by the implementation. Timestamps are expected to wrap around during the normal passage of time. It is recommended that an implementation pick a rate of advance and a bit width such that wrapping around occurs at most once every hour. Making the wrap time this long (or longer) makes timestamps more useful for several use cases.

- Checking for timeouts of protocol hello / keep-alive traffic that is on the order of seconds or minutes.
- If timestamps are placed into packets without converting them to other formats, then external data analysis systems using those timestamps will in many cases need to do so, e.g. to compare timestamps stored in packets by different PSA devices. These systems will need different formulas and/or parameters to perform this conversion for each wrap period, or to add extra external time references to the recorded data. The extra data required for accurate conversion is lower, and the likelihood of conversion mistakes is lower, if the timestamp values wrap less often.
- If timestamps are converted to other formats within a P4 program, it will need access to parameters that are likely to change every wrap time, e.g. at least a "base value" to add some calculated value to. A straightforward way to do this requires the control plane to update these values at least once or twice per timestamp wrap time.
- Programs that wish to use `(egress_timestamp - ingress_timestamp)` to calculate the queueing latency experienced by a packet need the wrap time to exceed the maximum queueing latency.

Examples of the number of bits required for wrap times of at least one hour:

- A 32-bit timestamp advancing by 1 per microsecond takes 1.19 hours to wrap.
- A 42-bit timestamp advancing by 1 per nanosecond takes 1.22 hours to wrap.

A PSA implementation is not required to implement time synchronization, e.g. via PTP[4] or NTP[5].

TBD: This text has been written assuming that it is more important for timestamps to be increasing at a constant rate, with no sudden "jumps" due to time synchronization events. Is this what people want from timestamps?

TBD: Some time synchronization methods avoid sudden "jumps" by temporarily speeding up or slowing down the rate of increase by a small percentage, until the desired synchronization is achieved. (TBD: which ones? citation?). Would anyone mind if PSA implementations were allowed to do this with their timestamp values?

The control plane API excerpt below is intended to be added as part of the P4Runtime API[1].

TBD: Antonin Bas suggests exposing 'switch capabilities' type of information like that shown below not via the P4Runtime API, but instead via Openconfig data models. Ask him for a more

[2] http://p4.org/p4/inband-network-telemetry
[3] https://en.wikipedia.org/wiki/Dynamic_frequency_scaling
[4] https://en.wikipedia.org/wiki/Precision_Time_Protocol
[5] https://en.wikipedia.org/wiki/Network_Time_Protocol
[1] The P4Runtime API, defined as a Google Protocol Buffer `.proto` file, can be found at https://github.com/p4lang/PI/blob/master/proto/p4/p4runtime.proto

appropriate way to document this in the PSA spec.

```
// The TimestampInfo and Timestamp messages should be added to the
// "oneof" inside of message "Entity".

// TimestampInfo is only intended to be read.  Attempts to update this
// entity have no effect, and should return an error status that the
// entity is read only.

message TimestampInfo {
  // The number of bits in the device's `Timestamp_t` type.
  uint32 size_in_bits = 1;
  // The timestamp value of this device increments
  // `increments_per_period` times every `period_in_seconds` seconds.
  uint64 increments_per_period = 2;
  uint64 period_in_seconds = 3;
}

// The timestamp value can be read or written.  Note that if there are
// already timestamp values stored in tables or `Register` instances,
// they will not be updated as a result of writing this timestamp
// value.  Writing the device timestamp is intended only for
// initialization and testing.

message Timestamp {
  bytes value = 1;
}
```

For every packet `P` that is processed by ingress and then egress, with the minimum possible latency in the packet buffer, it is guaranteed that the `egress_timestamp` value for that packet will be the same as, or slightly larger than, the `ingress_timestamp` value that the packet was assigned on ingress. By "slightly larger than", we mean that the difference `(egress_timestamp - ingress_timestamp)` should be a reasonably accurate estimate of this minimum possible latency through the packet buffer, perhaps truncated down to 0 if timestamps advance more slowly than this minimum latency.

Consider two packets such that at the same time (e.g. the same clock cycle), one is assigned its value of `ingress_timestamp` near the time it begins parsing, and the other is assigned its value of `egress_timestamp` near the time that it begins its egress processing. It is allowed that these timestamps differ by a few tens of nanoseconds (or by one "tick" of the timestamp, if one tick is larger than that time), due to practical difficulties in making them always equal.

Recall that the binary operators `+` and `-` on the `bit<W>` type in P4 are defined to perform wrap-around unsigned arithmetic. Thus even if a timestamp value wraps around from its maximum value back to 0, you can always calculate the number of ticks that have elapsed from timestamp $t1$ until timestamp $t2$ using the expression $(t2-t1)$ (if more than 2^W ticks have elapsed, there will be aliasing of the result). For example, if timestamps were $W >= 4$ bits in size, $t1 = 2^W - 5$, and $t2 = 3$, then $(t2 - t1) = 8$.

It is sometimes useful to minimize storage costs by discarding some bits of a timestamp value in a P4 program for use cases that do not need the full wrap time or precision. For example, an application that only needs to detect protocol timeouts with an accuracy of 1 second can discard the least significant bits of a timestamp that change more often than every 1 second.

Another example is an application that needed full precision of the least significant bits of a timestamp, but the combination of the control plane and P4 program are designed to examine all entries of a `Register` array where these partial timestamps are stored more often than once every 5 seconds, to prevent wrapping. In that case, the P4 program could discard the most significant bits of the timestamp so that the remaining bits wrap every 8 seconds, and store those partial timestamps in the `Register` instance.

5. Programmable blocks

The following declarations provide a template for the programmable blocks in the PSA. The P4 programmer is responsible for implementing controls that match these interfaces and instantiate them in a package definition.

It uses the same user-defined metadata type `IM` and header type `IH` for all ingress parsers and control blocks. The egress parser and control blocks can use the same types for those things, or different types, as the P4 program author wishes.

```
parser IngressParser<H, M>(packet_in buffer,
                           out H parsed_hdr,
                           inout M user_meta,
                           in psa_ingress_parser_input_metadata_t istd,
                           out psa_parser_output_metadata_t ostd);

control Ingress<H, M>(inout H hdr, inout M user_meta,
                      PacketReplicationEngine pre,
                      in    psa_ingress_input_metadata_t  istd,
                      inout psa_ingress_output_metadata_t ostd);

parser EgressParser<H, M>(packet_in buffer,
                          out H parsed_hdr,
                          inout M user_meta,
                          in psa_egress_parser_input_metadata_t istd,
                          out psa_parser_output_metadata_t ostd);

control Egress<H, M>(inout H hdr, inout M user_meta,
                     BufferingQueueingEngine bqe,
                     in    psa_egress_input_metadata_t  istd,
                     inout psa_egress_output_metadata_t ostd);

control Deparser<H, M>(packet_out buffer, inout H hdr, in M user_meta);

package PSA_Switch<IH, IM, EH, EM>(IngressParser<IH, IM> ip,
                                   Ingress<IH, IM> ig,
                                   Deparser<IH, IM> id,
                                   EgressParser<EH, EM> ep,
                                   Egress<EH, EM> eg,
                                   Deparser<EH, EM> ed);
```